

Trabalho N1 – Tipo 3 (versão 3) – Valor: 10%

O objetivo deste trabalho é compreender como o ciclo de instrução da Máquina de von Neumann. Para isso, cada **dupla de alunos** deverá implementar um “simulador” de uma CPU com o conjunto de instruções (ISA – *Instruction Set Architecture*), o conjunto de registradores arquiteturais e o formato de instrução listados abaixo. Esse simulador deve mostrar o conteúdo dos registradores no fim de cada ciclo de máquina, quando haverá uma pausa até apertar uma tecla para iniciar o próximo ciclo.

O prazo de entrega é 13/04/2024, sábado, às 23:59 via Moodle. Como de praxe, **cópia ou semelhança nas implementações serão punidas com nota zero** para todas as duplas em que isso for detectado. Passar o seu código para o colega é antiético, antiprofissional e está atrapalhando no desenvolvimento dele. Além disso, corre-se o risco da punição e, no fim do semestre, todo ponto faz a diferença entre ser aprovado ou não. Ainda, **duplas que não tiverem tirado nenhuma dúvida comigo até a entrega do trabalho terão a nota zero atribuída à atividade**. Isso é para evitar alunos que simplesmente aparecem com o trabalho pronto sem nunca ter tirado dúvidas, o que é extremamente improvável.

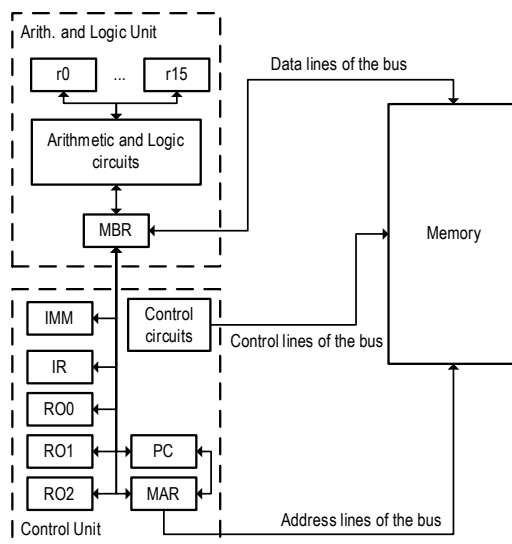
Os entregáveis são: todos os códigos-fonte necessários para a compilação e o programa compilado, assim como instruções de compilação e de uso do programa.

Ele deve ser implementado em linguagem C. Não é C++ e, sim, C. Portanto, sem orientação a objeto, pois só complica desnecessariamente o desenvolvimento deste trabalho em específico. Para facilitar, **a memória deve ser um vetor de palavras de oito bits** de 154 posições, o que, consequentemente, totalizará essa mesma quantidade em bytes. Em outras palavras, `unsigned char memoria[154]`. **Não serão aceitas** implementações onde os registradores sejam um vetor de caracteres ou a memória, uma matriz de caracteres. O programa deve ser colocado na memória na forma binária respeitando os formatos das palavras de instrução abaixo. Outra característica dessa memória é que ela usa um barramento com oito linhas de dados – ou seja, um barramento de oito bits. Sendo assim, **todas as transferências entre MBR e a memória devem ser de byte em byte**.

O simulador **deve** possuir uma maneira de ler um arquivo texto para carregar a memória com instruções e dados como apresentado na última página. É importante ressaltar, porém, que o simulador não precisa ser “bonito”, mas precisa ser fácil de usar.

Além disso, a CPU a ser implementada processa apenas números inteiros contidos em palavras de 32 bits e, portanto, não há nenhuma operação com ponto-flutuante, BCD ou números inteiros de outros tamanhos. Ademais, não é necessário implementar nenhuma representação de aritmética sinalizada. Embora o formato de instrução permita endereçar até $2^{23} = 8.388.608$ palavras de oito bits na memória, a memória possui apenas 154 endereços, como apresentado acima, abrangendo os endereços de 0 (0x0) a 153 (0x99).

De maneira muito simplificada, o diagrama da CPU com os registradores arquiteturais é:



E a função de cada registrador arquitetural é:

1. **MBR** – *Memory Buffer Register* – contém a palavra a ser armazenada na memória. Também é o registrador usado para receber uma palavra lida da memória. Todo o tráfego de e para a memória RAM deve passar pelo MBR. Deve ser implementado como uma variável de 32 bits (`unsigned int mbr`);
2. **MAR** – *Memory Address Register* – especifica o endereço de memória a ser lida da ou escrita na memória. Todo endereço de memória deve ser indicado nesse registrador antes da execução da instrução. Teoricamente, ele deveria ter o tamanho de 23 bits, mas como não existe variável de 23 bits na linguagem C, ele deve ser implementado como uma variável de 32 bits (`unsigned int mar`);
3. **IR** – *Instruction Register* – contém o *opcode* da instrução a ser executada. Teoricamente, ele deveria ter o tamanho de cinco bits, mas como não existe variável de cinco bits na linguagem C, ele deve ser implementado como uma variável de oito bits (`unsigned char ir`);
4. **RO0** – *Register Operand 0* – contém o endereço do primeiro operando registrador da instrução. Teoricamente, ele deveria ter o tamanho de quatro bits, mas como não existe variável de quatro bits na linguagem C, ele deve ser implementado como uma variável de oito bits (`unsigned char ro0`);
5. **RO1** – *Register Operand 1* – contém o endereço do segundo operando registrador da instrução. Teoricamente, ele deveria ter o tamanho de quatro bits, mas como não existe variável de quatro bits na linguagem C, ele deve ser implementado como uma variável de oito bits (`unsigned char ro1`);
6. **RO2** – *Register Operand 2* – contém o endereço do terceiro operando registrador da instrução. Teoricamente, ele deveria ter o tamanho de quatro bits, mas como não existe variável de quatro bits na linguagem C, ele deve ser implementado como uma variável de oito bits (`unsigned char ro2`);
7. **IMM** – *Immediate* – contém o operando imediato da instrução. Teoricamente, ele deveria ter o tamanho de 23 bits, mas como não existe variável de 23 bits na linguagem C, ele deve ser implementado como uma variável de 32 bits (`unsigned int imm`);
8. **PC** – *Program Counter* – contém o endereço da próxima palavra de instrução a ser buscada na memória. Caso não haja nenhum desvio, `halt` ou `nop`, o PC deve ser incrementado em cada ciclo de instrução. Deve ser implementado como uma variável de 32 bits (`unsigned int pc`);
9. **E, L e G** – registradores internos que armazenam as *flags* 'equal to', 'lower than' e 'greater than'. Cada uma delas contém um bit indicando se o conteúdo do primeiro operando registrador é, ao ser comparado pela instrução `cmp`, respectivamente 1) igual a, 2) menor do que ou 3) maior do que o conteúdo do segundo operando registrador. Como não há maneira de implementar variáveis de um bit, devem ser implementados como variáveis de oito bits (`unsigned char e, unsigned char l, unsigned char g`);
10. **r0 a r15** – registradores de propósito-geral (GPRs) utilizados para manter temporariamente os operandos na ALU. Devem ser implementados como um vetor com posições de 32 bits (`unsigned int reg[16]`). Esses registradores são codificados nos campos de instrução `reg0`, `reg1` e `reg2` da seguinte forma:



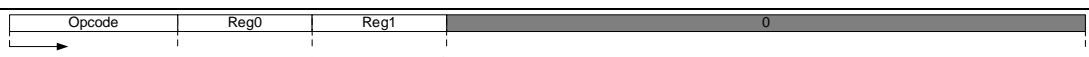
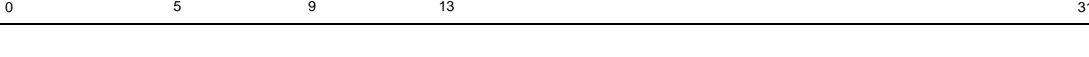
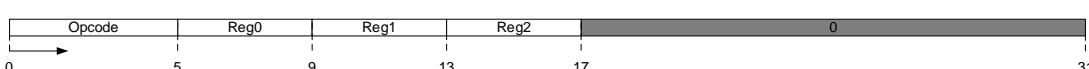
Reg.	Sequência de bits	Reg.	Sequência de bits
r0	0000	r8	1000
r1	0001	r9	1001
r2	0010	r10	1010
r3	0011	r11	1011
r4	0100	r12	1100
r5	0101	r13	1101
r6	0110	r14	1110
r7	0111	r15	1111


Finalmente, o conjunto de instruções é:

Mnemônico	Opcode	Descrição
halt	0b000000	HALT: o processador não faz nada. Em outras palavras, nenhum registrador tem o seu valor alterado durante a execução de <code>halt</code> . Deve-se colocar no fim do programa.

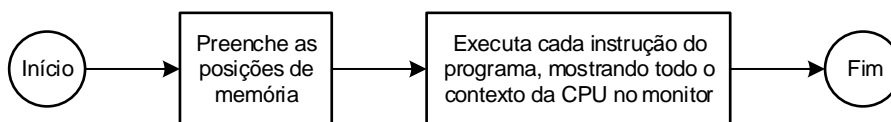
Mnemônico	Opcode	Descrição
nop	0b00001	NO OPERATION: O PC é incrementado, mas <u>nenhum</u> outro registrador tem seu valor alterado durante a execução de nop.
not rX	0b00010	LOGICAL-NOT ON REGISTER: $rX = !rX$
movr rX, rY	0b00011	MOVE REGISTER: $rX = rY$
cmp rX, rY	0b00100	COMPARE REGISTER: compara a palavra no registrador X com a palavra no registrador Y e preenche os registradores internos E, L e G os valores fazendo sequencialmente os seguintes testes: 1. Se $rX = rY$, então $E = 1$; senão $E = 0$; 2. Se $rX < rY$, então $L = 1$; senão $L = 0$; 3. Se $rX > rY$, então $G = 1$; senão $G = 0$.
ldbo rX, rY, M[Z]	0b00101	LOAD VIA BASE+OFFSET: $rX = * (M[Z] + rY)$
stbo rX, rY, M[Z]	0b00110	STORE VIA BASE+OFFSET: $* (M[Z] + rY) = rX$
add rX, rY, rZ	0b00111	ADD REGISTER: $rX = rY + rZ$
sub rX, rY, rZ	0b01000	SUBTRACT REGISTER: $rX = rY - rZ$
mul rX, rY, rZ	0b01001	MULTIPLY REGISTER: $rX = rY \times rZ$
div rX, rY, rZ	0b01010	DIVIDE REGISTER: $rX = rY \div rZ$
and rX, rY, rZ	0b01011	LOGICAL-AND ON REGISTER: $rX = rY \& rZ$
or rX, rY, rZ	0b01100	LOGICAL-OR ON REGISTER: $rX = rY rZ$
xor rX, rY, rZ	0b01101	LOGICAL-XOR ON REGISTER: $rX = rY \wedge rZ$
ld rX, M[Y]	0b01110	LOAD: carrega para o registrador X uma palavra da memória de 32 bits que se inicia no endereço Y.
st rX, M[Y]	0b01111	STORE: armazena uma palavra de 32 bits que começa a partir do endereço de memória Y o conteúdo do registrador X.
movil rX, imm	0b10000	MOVE IMMEDIATE TO THE LOWER HALF OF THE REGISTER: zera o registrador X e move os dezesseis bits menos significativos (0:15) do imediato para a parte inferior (0:15) do registrador X.
movih rX, imm	0b10001	MOVE IMMEDIATE TO THE HIGHER HALF OF THE REGISTER: move os dezesseis bits menos significativos (0:15) do imediato para a parte superior (16:31) do registrador X, enquanto os bits menos significativos do registrador X são mantidos intactos.
addi rX, imm	0b10010	ADD IMMEDIATE: $rX = rX + IMM$
subi rX, imm	0b10011	SUBTRACT IMMEDIATE: $rX = rX - IMM$
muli rX, imm	0b10100	MULTIPLY IMMEDIATE: $rX = rX \times IMM$
divi rX, imm	0b10101	DIVIDE IMMEDIATE: $rX = rX \div IMM$
lsh rX, imm	0b10110	LEFT SHIFT: desloca a palavra no registrador X em IMM bits à esquerda.
rsh rX, imm	0b10111	RIGHT SHIFT: desloca a palavra no registrador X em IMM bits à direita.
je M[X]	0b11000	JUMP IF EQUAL TO: muda o registrador PC para o endereço de memória X caso $E = 1$
jne M[X]	0b11001	JUMP IF NOT EQUAL TO: muda o registrador PC para o endereço de memória X caso $E = 0$.
jl M[X]	0b11010	JUMP IF LOWER THAN: muda o registrador PC para o endereço de memória X caso $L = 1$.
jle M[X]	0b11011	JUMP IF LOWER THAN OR EQUAL TO: muda o registrador PC para o endereço de memória X caso $E = 1$ ou $L = 1$.
jg M[X]	0b11100	JUMP IF GREATER THAN: muda o registrador PC para o endereço de memória X caso $G = 1$.
jge M[X]	0b11101	JUMP IF GREATER THAN OR EQUAL TO: muda o registrador PC para o endereço de memória X caso $E = 1$ ou $G = 1$.
jmp M[X]	0b11110	JUMP: muda o registrador PC para o endereço de memória X.

As instruções são codificadas da seguinte forma:

Mnemônico	Formato da palavra de instrução
hlt nop	
not rX	
movr rX, rY	
cmp rX, rY	
add rX, rY, rZ	
sub rX, rY, rZ	
mul rX, rY, rZ	
div rX, rY, rZ	
and rX, rY, rZ	
or rX, rY, rZ	
xor rX, rY, rZ	

Mnemônico	Formato da palavra de instrução
ld rX, M[Y]	
st rX, M[Y]	
movl rX, imm	
movih rX, imm	
addi rX, imm	
subi rX, imm	
muli rX, imm	
divi rX, imm	
lsh rX, imm	
rsh rX, imm	
je M[X]	
jne M[X]	
jl M[X]	
jle M[X]	
jg M[X]	
jge M[X]	
jmp M[X]	
ldbo rX, rY, M[Z]	
stbo rX, rY, M[Z]	

Observe que a CPU simulada deverá ser capaz de executar **qualquer** programa que for adicionado à memória. O programa deve ser colocado na memória na forma binária respeitando os formatos das palavras de instrução ilustradas na tabela acima. Dessa forma, o funcionamento do programa deve seguir a seguinte sequência:



Para a primeira etapa – carregar a memória – o simulador deve ler um arquivo texto como apresentado na última página. Essa etapa deve continuar enquanto o programa não encontrar a linha contendo 'hlt' (sem os apóstrofes). A conversão preencherá a memória em uma maneira semelhante à observada abaixo.

End. mem.	Conteúdo	Conteúdo codificado em binário	Conteúdo codificado em hexadecimal
0x0	ld r0, 18	0111 0000 0000 0000 0000 0000 0001 1000	70 00 00 18
0x4	ld r1, 1C	0111 0000 1000 0000 0000 0000 0001 1100	70 80 00 1C
0x8	addi r2, r0, r1	0011 1001 0000 0000 1000 0000 0000 0000	39 00 80 00
0xC	addi r0, 20	1001 0000 0000 0000 0000 0000 0010 0000	90 00 00 20
0x10	st r0, 20	0111 1000 0000 0000 0000 0000 0010 0000	78 00 00 20
0x14	hlt	0000 0000 0000 0000 0000 0000 0000 0000	00 00 00 00
0x18	15	0000 0000 0000 0000 0000 0000 0000 1111	00 00 00 0F
0x1C	8	0000 0000 0000 0000 0000 0000 0000 1000	00 00 00 08

Você pode observar na última coluna da tabela acima que os valores estão agrupados em pares de valores hexadecimais, onde cada um deles é um byte (ou o equivalente a uma posição de memória). Dessa forma, o mapa de memória fica assim (os endereços de memória e os seus conteúdos estão em hexadecimal):

Endereço	Conteúdo	Endereço	Conteúdo	Endereço	Conteúdo	Endereço	Conteúdo
0	70	A	80	14	00	1E	00
1	00	B	00	15	00	1F	08
2	00	C	90	16	00	20	...
3	18	D	00	17	00	21	...
4	70	E	00	18	00	22	...
5	80	F	20	19	00	23	...
6	00	10	78	1A	00	24	...
7	1C	11	00	1B	0F	25	...
8	39	12	00	1C	00	26	...
9	00	13	20	1D	00	27	...

O que estará na memória é o que está nessa tabela.

Na segunda etapa – que é exibir o funcionamento da CPU executando o programa – o trabalho deve exibir algo assim (considerando que todos os conteúdos de registradores e da memória, incluindo os seus endereços, estão em hexadecimal):

```
CPU:
R0: 0xFFFFFFFF R1: 0xFFFFFFFF R2: 0xFFFFFFFF R3: 0xFFFFFFFF
R4: 0xFFFFFFFF R5: 0xFFFFFFFF R6: 0xFFFFFFFF R7: 0xFFFFFFFF
R8: 0xFFFFFFFF R9: 0xFFFFFFFF R10: 0xFFFFFFFF R11: 0xFFFFFFFF
R12: 0xFFFFFFFF R13: 0xFFFFFFFF R14: 0xFFFFFFFF R15: 0xFFFFFFFF
MBR: 0xFFFFFFFF MAR: 0xFFFFFFFF IMM: 0xFFFFFFFF PC: 0xFFFFFFFF
IR: 0xFF R00: 0xF R01: 0xF R02: 0xF
E: 0xF L: 0xF G: 0xF
```

```
Memória:
00: 0xFF 01: 0xFF 02: 0xFF 03: 0xFF
... .. 99: 0xFF
```

Pressione uma tecla para iniciar o próximo ciclo de máquina ou aperte CTRL+C para finalizar a execução do trabalho.

Deve-se exibir o conteúdo dos registradores e da memória ao fim de cada ciclo de máquina, com o usuário devendo pressionar uma tecla para iniciar a execução do próximo ciclo.

Para ajudar no desenvolvimento do trabalho, teste-o com os dois programas abaixo, considerando que eles já estão no formato esperado de arquivo esperado de arquivo a ser lido pelo trabalho. Todos os dados, endereços e imediatos estão representados em hexadecimal. Observe que as linhas que estão no formato endereço;instrução/dado;palavra de instrução ou palavra de dado. Em todo caso, o endereço inicial de memória que aquela instrução ou dado ocupará é dado em hexadecimal, assim como os valores numéricos nas instruções. Além disso, observe que as instruções e os dados ocupam quatro bytes.

$A = 32 + 3 \times \frac{4}{5 - 3}$	$A = \sum_{1}^{10} \frac{10}{5} + 3 \times (2 - 1)$
<pre>0;i;ld r0, 86 4;i;ld r1, 8a 8;i;ld r2, 8e c;i;ld r3, 92 10;i;ld r4, 96 14;i;sub r5, r3, r4 18;i;div r6, r2, r5 1c;i;mul r7, r6, r1 20;i;add r8, r0, r7 24;i;st r8, 82 28;i;hlt 86;d;20 8a;d;3 8e;d;4 92;d;5 96;d;3</pre>	<pre>0;i;ld r0, 7a 4;i;ld r1, 7e 8;i;ld r2, 82 c;i;ld r3, 86 10;i;ld r4, 8a 14;i;ld r5, 8e 18;i;ld r6, 92 1c;i;ld r7, 96 20;i;div r8, r3, r4 24;i;sub r9, r6, r7 28;i;mul r9, r9, r5 2c;i;add r9, r9, r8 30;i;add r0, r0, r9 34;i;addi r1, 1 38;i;cmp r1, r2 3c;i;jle 30 40;i;st r0, 7a 44;i;st r1, 7e 48;i;hlt 7a;d;0 7e;d;1 82;d;a 86;d;a 8a;d;5 8e;d;3 92;d;2 96;d;1</pre>